7 COMMON AGILE TECHNIQUES

7.1 STORIES AND BACKLOG REFINEMENT

Stories define what is required from the team by the customer and stakeholders (see Chapter 5). A story is not a detailed specification of a requirement; rather, it is a token or reminder for the team that a feature (or anything else that is value-add – for example, a story driving the creation of a security standard) needs to be delivered.

Stories should include:

- **WHO** wants a feature. It is good practice to write the 'Who' statement in the stories from the perspective of an 'Agile persona'. An Agile persona is any person or group (a 'user') who will interact with the features being created. The reason that stories are written from the perspective of an Agile persona is that it facilitates creation of stories that make sense to stakeholders and the customer.
- **WHAT** feature they want.
- **WHY** they want the feature. This may be tied into a business case if one exists.
- **ACCEPTANCE CRITERIA.** This is normally a list of questions, scenarios or examples that enable the customer to sign off the story as 'done' (see below for more information and Section 10.2.1 for more information about 'done').

Stories usually comprise a story card (a physical or virtual piece of card that describes the information above; see Figure 7.1), any conversation(s) between the customer and team (that may be added to the card) and, after the feature has been delivered, confirmation from the customer that the delivery meets acceptance criteria. A typically used format and syntax for stories is:

- **As a** (the 'who') …
- **I want** (the 'what') …
- **So that** (the 'why') …
- **Acceptance criteria**

Box 7.1 An example of story writing

This is an example of how a story might be written, relating to an Agile persona: 'Mike – the Business Development Manager'.

As a – Business Development Manager

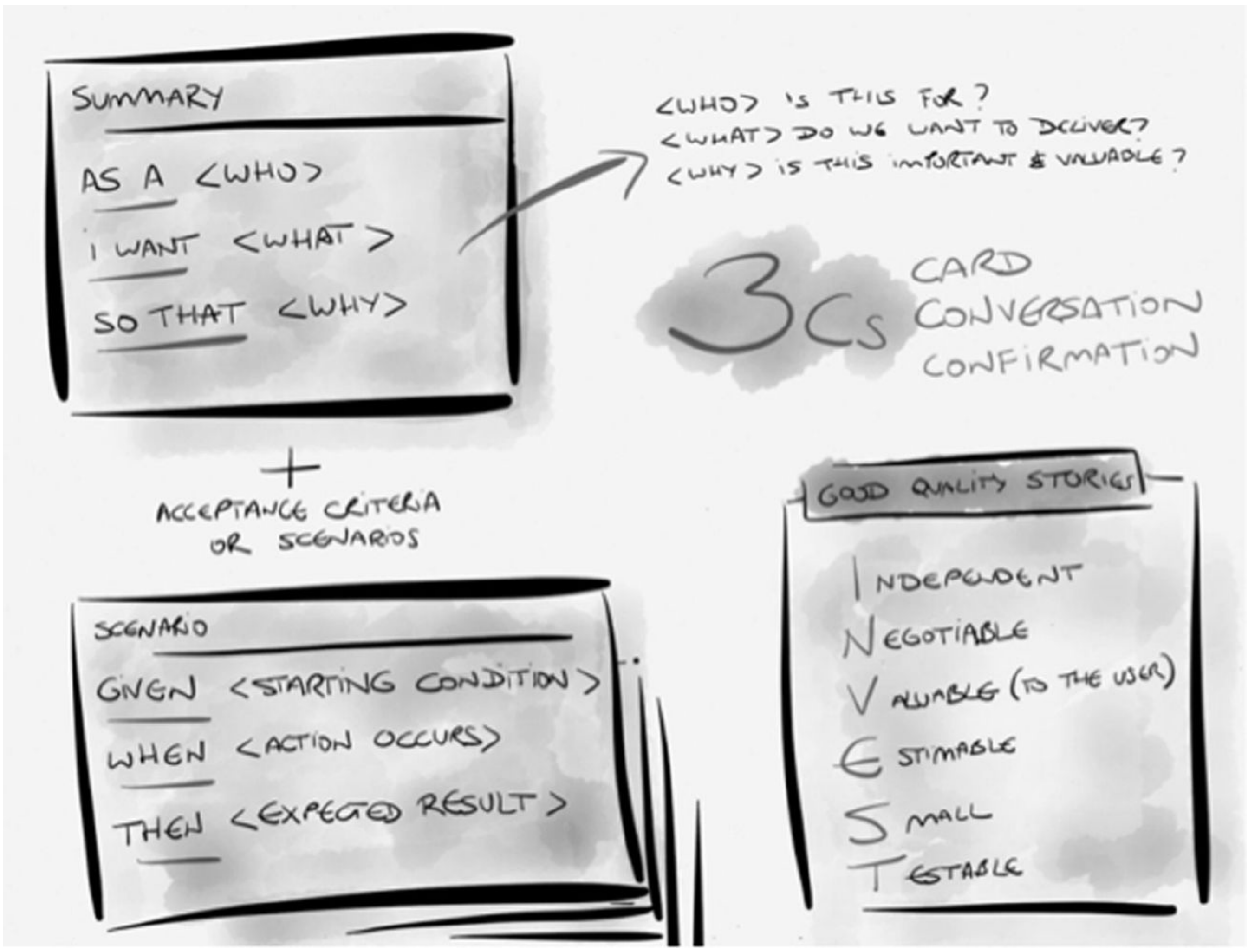
I want – the ability to identify all people who have registered or re-registered on our system in the last 3 months

So that – I can send focused marketing material to those people

Acceptance criteria –

- Can I identify all people who have registered or re-registered in the last 3 months from today's date?
- Can I identify basic demographics (name, age, email address) relating to those people?
- Is it clear what the core preferences, via site usage, of those people are?
- Do I know how many times these people have logged on in the last 3 months?
- Do I know how much those people have spent with us each month?
- Do I know what products these people have ordered each month?
- Am I prevented from seeing people who registered or re-registered outside the 3 month timeline? (this is an example of a 'negative' acceptance criteria written as a 'positive' question).

Figure 7.1 Story cards



Stories can be of any size, and they tend to be refined as the product is developed, down from very large size (known as 'coarse-grained stories'), which could be months of effort in size to very small (known as 'fine-grained stories'), which are typically 1–5 days' effort in size.

Stories are placed in a backlog, which is essentially a ‘to-do’ list where stories are arranged in an agreed delivery sequence. Stories in a backlog are continually refined throughout the whole lifetime of the product. This lifetime could be anything from a couple of months (e.g. a marketing campaign) to many decades (e.g. a banking system). The customer and the team refine stories on an ongoing basis throughout the whole lifetime of the product. It is expected that stories will change and be refined, because the team are working in an environment of high variability (see Section 2.2).

A common mistake made by teams is that they write technical stories that do not make sense to the customer. This inhibits agility as the customer cannot provide a priority/sequence in which to deliver the feature as they do not understand the story; and it also inhibits collaboration between the team, customer and stakeholders.

A good way to understand the characteristics of good stories is the acronym ‘INVEST’. It stands for:

Independent Stories should be deliverable independently of each other. This is generally feasible at the coarse-grained level but can become more difficult as stories become more fine-grained. Creating independent stories also enables the team and customer to inject small stories into the backlog that can be delivered in timescales aligned to iterations/sprints (‘Feature Injection’; Matts, 2013).

Negotiable A story is not a detailed specification of requirements; rather it is something that will be refined over time, and is negotiable up until the point that the story is planned within a sprint.

Valuable It is essential that the value of a story is understood by the customer. Only if the customer can identify the value of a story can it be ordered within the backlog. This means a common, non-jargon-laden language for all stories is fundamentally important.

Estimable The story is the unit against which plans and estimates are created. Stories must be understood by the team as they are responsible for creating estimates. Therefore it is essential that the team is involved in the refinement of stories, in cooperation with the customer and stakeholders. Only then will the team have a solid understanding of the story and be able to create realistic and achievable estimates.

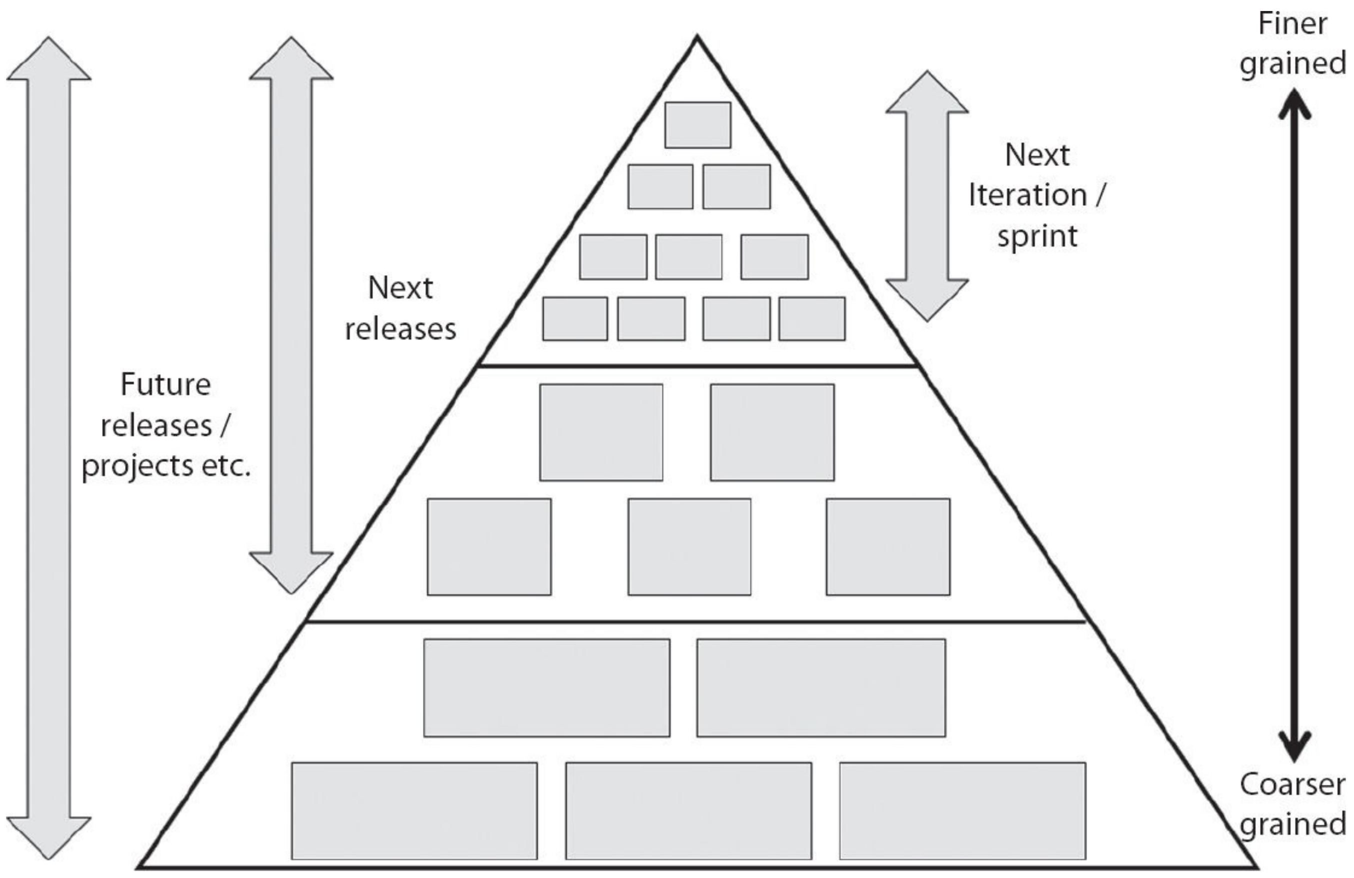
Small enough A common mistake when writing a story is to provide too much information too soon. Generally, it is recommended that stories are refined to the 1 to 5 days’ effort size just before the next iteration/sprint is planned.

Testable Stories must include testable acceptance criteria in order to achieve ‘done’ status (see Section 10.2.1). While acceptance criteria are not detailed tests (see Section 7.1.2), they will be driving what the tests will be.

7.1.1 Planning pyramid

As mentioned above, it is good practice to keep stories independent of each other, and to have as few parent–child levels as possible. However, when delivering large, complex projects, there may be dependencies between stories, and a feature breakdown structure of parent–child stories may be required; this is known as a 'planning pyramid' (see Figure 7.2).

Figure 7.2 Planning pyramid



A planning pyramid contains both coarse-grained and fine-grained stories; the coarse-grained stories are being refined to be fine-grained stories as delivery progresses. If planning is required over a longer timescale than just the next sprint, stories will only be refined to the most coarse-grained level that enables this (see Section 7.2). When delivering a product that requires a delivery estimate, a simple end-to-end prototype would typically be created that gives a high-level description of the product to be delivered. This enables very high level planning and estimation, and the creation of a predicted end-date for the delivery. At all times, the core point to remember is that it is highly likely the product will change during the development cycle.

7.1.2 Scenario-based acceptance criteria

Previously, we described an example of how a story could be written from the perspective of an Agile persona, including a number of acceptance criteria. This style of writing acceptance criteria is normally associated with an Agile practice called 'Test Driven Development' (TDD – see Section 7.4).

In 2004 Dan North (North, 2006) suggested a thought experiment called 'BDD'

(Behaviour Driven Development – see Section 7.4) to complement TDD. Behaviour Driven Development is concerned with the behaviour of the system, which means that the acceptance criteria in BDD are usually expressed as scenarios. Here is a simple example of how to write acceptance criteria in BDD:

As a – generic bank customer

I want – the ability to withdraw cash from an ATM

So that – I don’t have to visit the branch continually to draw money

Acceptance criteria:

Scenario One: the bank account is in credit

Given

– the customer requires to draw cash from the ATM

When

– the customer enters their card to the machine

– and the bank account is in credit

Then

– debit bank account

– and update transaction statement

– and return card

– and dispense cash

Scenario Two: the bank account has hit overdraft limit

Given

– the customer requires to draw cash from the ATM

When

– the customer enters their card to the machine

– and the bank account has hit overdraft limit

Then

– return card

– and display message ‘overdraft limit reached’

– and offer customer other services

BDD is generally implemented on coarse-grained stories more suited to scenario-based acceptance criteria, while TDD is implemented on fine-grained stories where a question-driven approach is more suitable.

7.1.3 Backlog refinement and ‘spike’ stories

Stories are continually refined within the backlog throughout the whole lifetime of the product, which could be a couple of months (e.g. a marketing campaign), or could possibly be many decades (e.g. banking system).

This means that stories will continue to evolve from inception of the product through to decommissioning the backlog. In particular in complicated, complex or anarchic environments it is fundamental that stories evolve in line with the changing environment. This is unnecessarily difficult to do if the stories have too much detail, and/or are created too early within the delivery. Instead, stories should be refined on a 'just-in-time' basis for the next sprint (this idea aligns to a concept in Lean called 'last responsible moment' (see Section 9.2)).

A common mistake is that teams write and refine all or many of the stories that may be required before actually developing anything. This often happens when a team is being asked to give an estimate for the whole product delivery (see Section 7.2 for guidance on Agile estimating); however, if a team do this they are basically just doing a Waterfall delivery, having performed an 'analysis' stage by defining and refining all stories up front. In Agile deliveries, stories are refined continually on a 'just in time' basis.

Agile teams sometimes quote YAGNI as a good acronym to keep in mind when thinking about whether a feature should be added to a product or not. YAGNI stands for 'You Ain't Gonna Need It' and can be applied when deciding whether stories should be added to the backlog. For example, if the product is only going to be live for a short period of time, then the technical quality of the product may not need to be particularly robust.

A spike story (a term originally from eXtreme Programming (see Section 14.1)) is a story that drives technical or functional research effort or investigative work. Creating a spike story is different to refining stories in the backlog; backlog refinement is an ongoing process of analysis and design; spiking, on the other hand, is initiating a story-driven activity to investigate something specific.

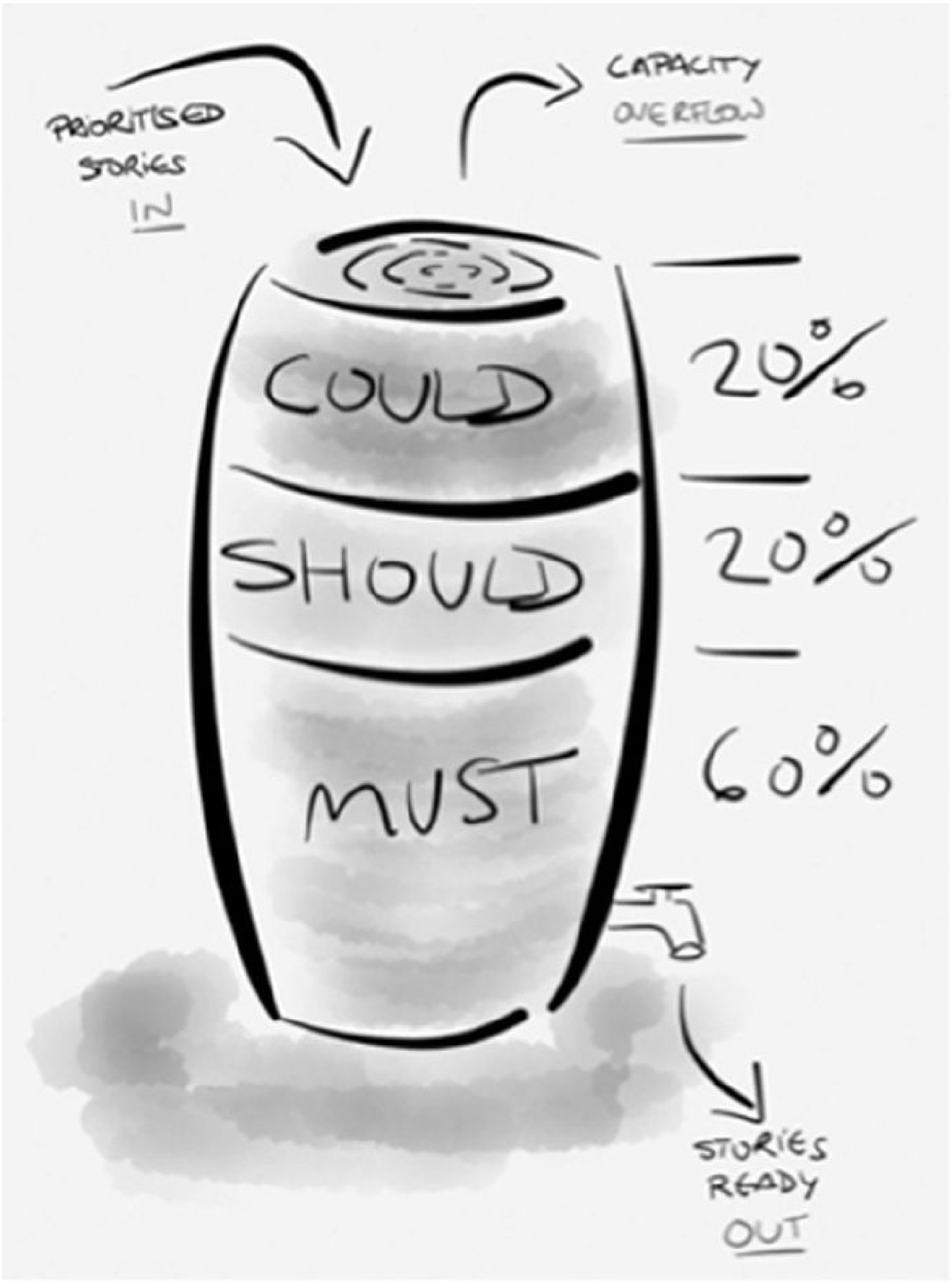
7.1.4 Prioritisation (with 'MSCW')

Prioritisation is key to all Agile frameworks because they all largely implement time-boxing (see Section 2.4.2.1). A time-box is a boxed period of time (e.g. a project, release or sprint/iteration) in which something will be delivered in a prioritised sequence.

There is a subtle difference between 'ordering' and 'prioritising' that is timescale-related. A backlog is a 'to-do' list of items, normally stories, which will be delivered in an agreed order throughout the lifetime of the product – from inception to decommissioning. Once some of the stories are planned into a project or release or iteration/sprint time-box, then prioritisation can be used to arrange them in a sequence within the time-box.

MSCW (pronounced MoSCoW) is a prioritisation technique that is the intellectual property of the DSDM Consortium (see Section 14.3), however, it is free to use and is used by many Agile teams (see Figure 7.3).

Figure 7.3 MSCW prioritisation



The MSCW acronym stands for the following features:

M – must have Sometimes also termed as the MVP (minimum viable product), or the MMFS (minimum marketable feature set). These are the stories that **must** be delivered within a particular time-box. Not delivering these stories and still delivering the product means the solution is non-viable, illegal or pointless.

S – should have A story that is very important within a time-box, and that will cause significant problems to the customer if not delivered, though the customer could still get value from the product if this feature is not in place.

C – could have A story that is very important within a time-box and may cause some problems to the customer if not delivered. However, the customer will still gain value from the product if a ‘could have’ feature is not in place.

W – won’t have this time It can be agreed between a customer and team that a particular story won’t be delivered in a particular time-box. This story might be added to a later time-box or removed completely from the backlog.

Once a time-box is finished, all missing stories should be reviewed and the priorities re-evaluated. A particular story may be a ‘must have’ feature for delivery at the end of a six-month project, but may only be a ‘should have’ feature for delivery at the end of a three-month release. The same feature may be a ‘could have’ feature for delivery within the first two-week sprint/iteration time-box.

It is important to understand that MSCW prioritisation is specifically designed for implementation within fixed time frames. If MSCW is implemented on a backlog without a time frame, it is very likely that a customer will define everything within the backlog as a ‘must have’.

The best way to think about MSCW prioritisation is ‘Within **this** specific time frame I must have/should have/could have/won’t have this feature.’

Another reason why MSCW prioritisation sometimes fails is if stories are defined at a very coarse-grained level. Again this will lead the customer to perceive them all as ‘must haves’ – for example, an e-commerce website must have a payment method, but there are opportunities to prioritise depending on the type, rigour, security and so on of payment method used.

One way of achieving MSCW prioritisation is to understand that there is more than one way to develop a feature. So the ‘must have’ story may be a basic implementation of a feature that enables the customer to gain value. Subsequently the ‘could have’ story can be a much more ‘gold plated’ version of the feature that the customer would ideally like. For example, the customer could require a date management service that manages all worldwide date formats, however, they must have a service that manages UK date format next month.

7.2 AGILE ESTIMATION

Agile estimates are forecasts of how many stories can be delivered within any time period (for example, in a project or a release iteration/sprint).

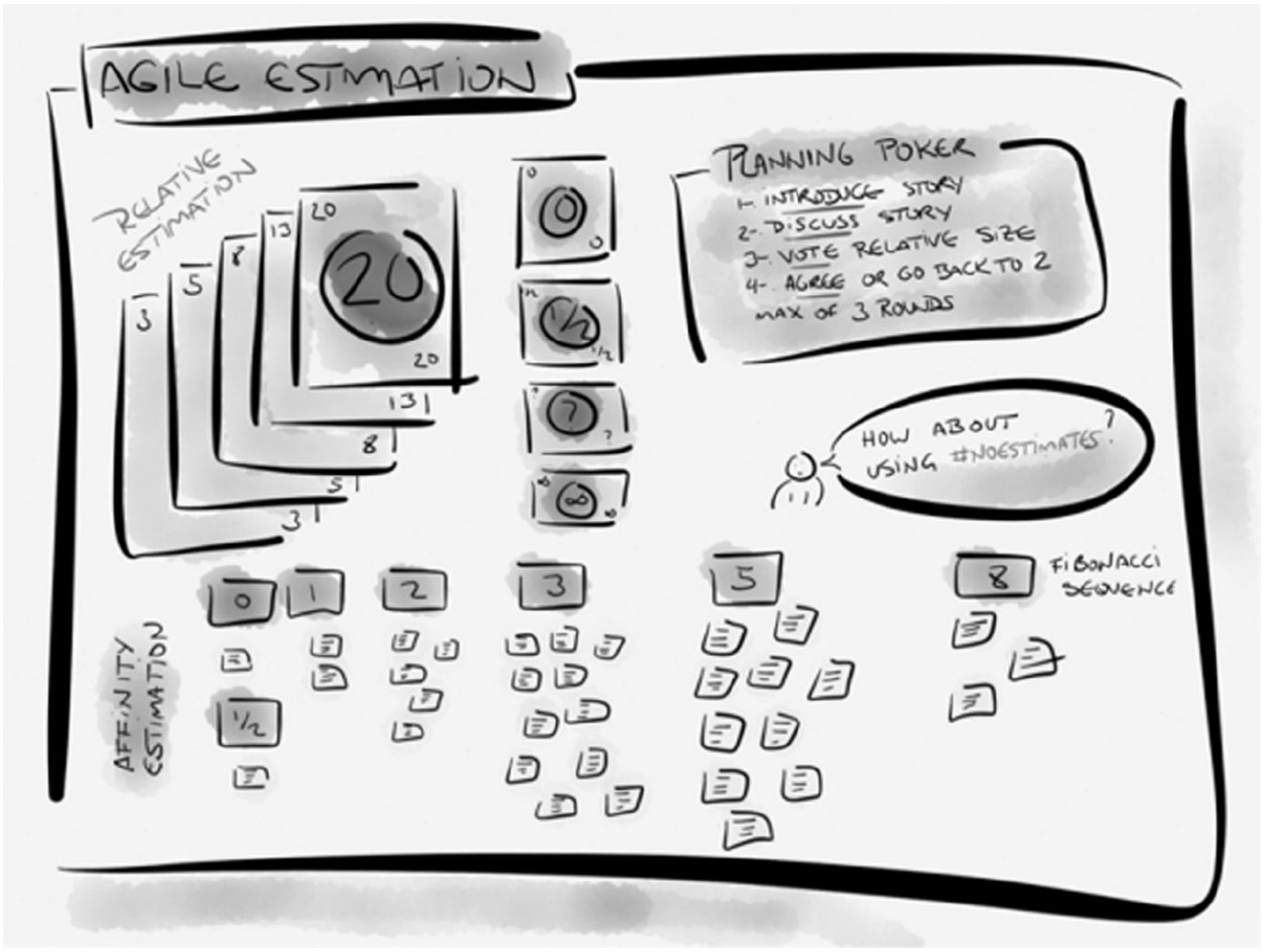
Story size and a team’s capacity to deliver stories (or ‘velocity’ – see below) are estimated in either ‘ideal days’ or ‘story points’. Story points and ideal days are specifically associated with ‘top-down’ estimation and planning (see Section 7.3.1; Figure 7.4).

It is important to remember that any estimates and plans in an Agile delivery tend to be made in uncertain and possibly volatile environments and therefore can only ever be baseline forecasts.

7.2.1 Ideal days

Ideal days are what project management frameworks call ‘productive time’; in essence the time that is allocated to deliver planned items (mainly stories) from the backlog. Time that is allocated to do other things such as answering telephone calls or emails, attending meetings and so on is often termed ‘non-productive time’ (although realistically it is not non-productive as this work and the associated time is also essential).

Figure 7.4 Agile estimation with story points



Many Agile teams tend to avoid estimates based on ideal days because once time is associated with coarse-grained stories teams may add contingency and ‘Parkinson’s law’ (‘work expands to fit time available’; Oxford Dictionary, 2014) may apply. Estimates based on time also carry the risk of too much up-front analysis and design being performed to ensure that the time-based estimate is ‘accurate’. Instead, many Agile teams use story points to create estimates because they are not directly related to time, meaning the issues described above are removed.

7.2.2 Story points and planning poker

Story points provide a way of relatively estimating story sizes, leading to a plan of how many stories can be delivered within a given time-box (for example, a project or release – see Section 7.3). Story points estimate the relative effort required to develop a particular story. As Mike Cohn, the originator of the concept of story points, says on one of his blogs, 'Story points are not about the complexity of developing a feature; they are about the effort required to develop a feature' (Cohn, n.d.).

Story points provide relative size estimates for stories. People use relative sizing all the time in their day-to-day lives: for example, they talk about one building being large and another building being small; or about one car being 'small-medium' and another car being 'medium-large'. Any new story can be relatively sized against the other stories already sized in the backlog.

Story points are generally determined by a team within a 'planning poker' workshop (Grenning, 2002), where they agree the relative effort (story points) required to deliver stories to 'done' status (see Section 10.2) on a scale from 0 to 100.

Planning poker typically uses a relative sizing based partly on a Fibonacci sequence as this describes fine- and coarse-grained differences between numbers. This numbering sequence aligns with the different relative sizes of stories. The relative differences between stories that are near to delivery will be fine-grained (because the stories have been refined), whereas the relative differences between stories that are further away from delivery will be coarse-grained.

Typical story-point sizes in planning poker are:

- The story is at 'done' status = 0
- Fine-grained sizings = ½ (xxs), 1 (xs), 2 (s), 3 (sm), 5 (m), 8 (ml), 13 (l) – the Fibonacci sequence
- Coarse-grained sizings = 20 (xl), 40 (xxl), 100 (xxxl)
- Not enough knowledge to estimate this story = ?
- The story is so big it's impossible to estimate or that the team do not have the capability to deliver the story at all = ∞ (infinity)

Using an architectural analogy: if a team identified the story 'Building a large shed' as requiring the least effort, they would size this story at (s) small (or '2' in the Fibonacci sequence). Other stories can now be sized relatively to this story. So, for example, the story 'Building a simple bedsit' may be relatively sized as (sm) 'small-medium' effort (5 in the Fibonacci sequence). This process continues until enough stories have been sized to enable planning.

7.3 AGILE PLANNING

Plans in an Agile delivery are specifically created to enable change. There are differing levels of plans in Agile frameworks, including portfolio, programme, project, release and iteration/sprint plans. This section will concentrate on release and iteration/sprint plans as these are fairly common across all Agile frameworks.

Many of the ideas that have become standard in relation to Agile estimating and planning are described in the book of the same name from Mike Cohn (2005).

7.3.1 Top-down and bottom-up planning

7.3.1.1 Top-down planning

Where variability is likely to be experienced it is highly risky to try and define a detailed plan, as there will be a significant overhead if it needs to be changed. Top-down plans (created by the team) are purposely quick and inexact and are especially suited to variable environments where things are likely to change.

Top-down planning is specifically associated with creating estimates for longer time frames, such as a release. When planning a release the accuracy of estimates will improve continually based on lessons learned throughout the life of the project (see Section 2.5).

Top-down planning can be performed with either ‘story points’ or ‘ideal days’ (see previous sections). Top-down planning is normally based on previous experience or existing reliable data.

An example could be assessing how long it will take to travel somewhere. If the journey has been done a few times before then top-down planning would use that previous experience and assume that the next time the journey will take an average of what it has taken previously.

7.3.1.2 Bottom-up planning

When a more detailed estimate is required, for example when committing to deliver in short iteration/sprint timescales of a few weeks, a bottom-up planning approach may be used and possibly calibrated with a top-down estimate.

In bottom-up planning, teams typically know which stories are likely to be delivered in the iteration/sprint based on top-down planning. The team then identify what capacity they have to deliver these stories; this is normally expressed as ‘total available hours’ within an iteration/sprint. The team then plan all the tasks that are required to get the stories to ‘done’ status, and estimate the hours needed to deliver the planned tasks; this is normally expressed as ‘total required hours’ in this iteration/sprint.

The ‘total required hours’ are then compared against the ‘total available hours’. If the figures differ, the team remove tasks until the required hours match the available hours. This may mean removing, replacing, adding or splitting some stories from the original top-down forecast.

7.3.2 Release planning and velocity

A release backlog is a subset of the overall backlog that relates to the stories that are forecast to be 'done' in a particular release. Release backlogs are created in a release planning activity at which the whole team, customer and possibly other stakeholders will be present.

To forecast the number of stories that can be done within a particular release backlog, teams must be able to estimate the size of stories using the same unit that they use to size story capacity within the release (ideal days or story points – see Section 7.2).

Once the team have estimated the relative size of the stories in the overall backlog, they then need to size the iterations/sprints within the release using the same unit (e.g. story points). To estimate how many story points can be achieved within an iteration/sprint, teams look to past historical evidence of how many story points they have actually achieved in previous sprints. This is called a 'velocity', i.e. the average number of story points that a team have been able to deliver across the last 1 to 5 iterations/sprints.

Once the team understand their velocity and the size of the stories in the backlog in the same unit (e.g. story points), they can plan the stories into the iterations/sprints in a release. Once all iterations/sprints in a release have stories assigned to them, a forecast plan of how many stories from the backlog can be achieved within a release can be created. At this point the release goal is defined. This is normally a subset of the stories planned into the release to give some slack for unforeseen circumstances.

If a release goal either becomes inaccurate (i.e. it does not deliver what the business now wants) or untenable (i.e. the forecasts were significantly wrong), release planning needs to be re-initiated and a new release goal agreed. This process is very similar at portfolio, programme and project levels.

It is important to keep in mind that release plans are baseline plans, and that they are likely to change as delivery progresses. A significant part of the organisational change to the Agile mindset (see Section 2.1) is therefore to ensure that stakeholders understand that release plans are designed to change, rather than being commitment plans (this is why they are top-down plans. If stakeholders force any plan at a level above the iteration/sprint plan to be a commitment plan, it will typically only achieve the delivery of the wrong product, because it does not allow for the plan to respond to changing business needs.

7.3.3 Iteration/sprint planning

Iteration/sprint planning is usually performed in two parts, often named 'Planning part 1' and 'Planning part 2'.

7.3.3.1 Iteration/sprint planning part 1

Sprint planning part 1 is an example of top-down planning (see above), and is performed in the same way as release planning – i.e. the team plan the number of stories to be delivered within an iteration/sprint based on story-point size and velocity.

7.3.3.2 Iteration/sprint planning part 2

Sprint planning part 2 is an example of bottom-up planning (see Section 7.3.1.2). When the forecast number of stories that can be delivered from top-down and bottom-up estimates match, this is likely to be a robust estimate that the team are able to commit to. If the figures do not match then the team will need to ascertain the reasons for the discrepancy to the point where they believe they have arrived at a robust estimate for the iteration/sprint.

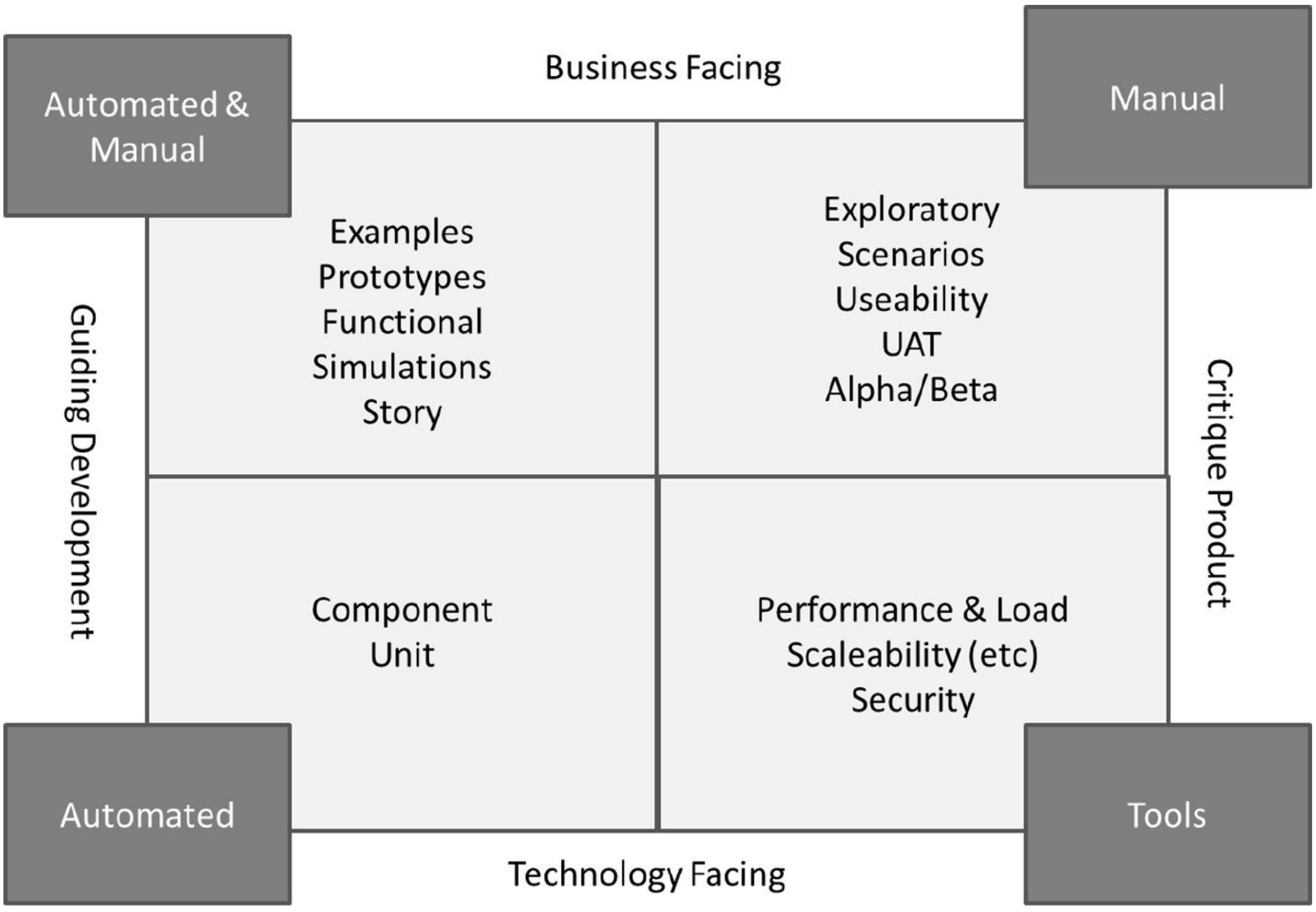
7.4 AGILE TESTING

A core principle of Agile quality control is that testing (validation) is integrated throughout the entire lifecycle. All types of testing need to be implemented early and continuously and should never be left to the end of a release period, as significant defects found at that point can seriously derail delivery or quality of the product.

Many Agile frameworks align to the concepts described in the book *Agile Testing* by Lisa Crispin and Janet Gregory (Crispin and Gregory, 2009) in their approach to quality control and testing. The book presents an evolution of the 'Agile Testing Quadrants', originally defined by Brian Marick.

Figure 7.5 proposes a simple-to-understand model of what is included in Agile testing. It gives guidance on the types of testing that may be used in an Agile delivery, as well as what the testing should focus on and how it should be delivered.

Figure 7.5 Agile testing quadrants



There are a number of testing-based development practices that are typically used in Agile deliveries. These are described in the following sections.

7.4.1 Agile testing practices

7.4.1.1 TFD (Test First Development)

As the name suggests this practice means that tests are written before any development or coding to meet story acceptance criteria happens. When a story is to be developed, team members with analysis and design, build and test skills (an approach sometimes called ‘The Three Amigos’ (Hewitt, 2013)) get together and develop the tests, edge cases and so on just prior to delivery.

Where it is practical the customer will also be involved in creating tests so that they have confidence that the tests will prove that the acceptance criteria are met.

Once the tests have been written, a test–build cycle (test, then build, then test, then build etc., up to the point all tests are passed) is implemented until any and all bugs associated with the build have been fixed and the customer is happy to sign the story off as ‘done’.

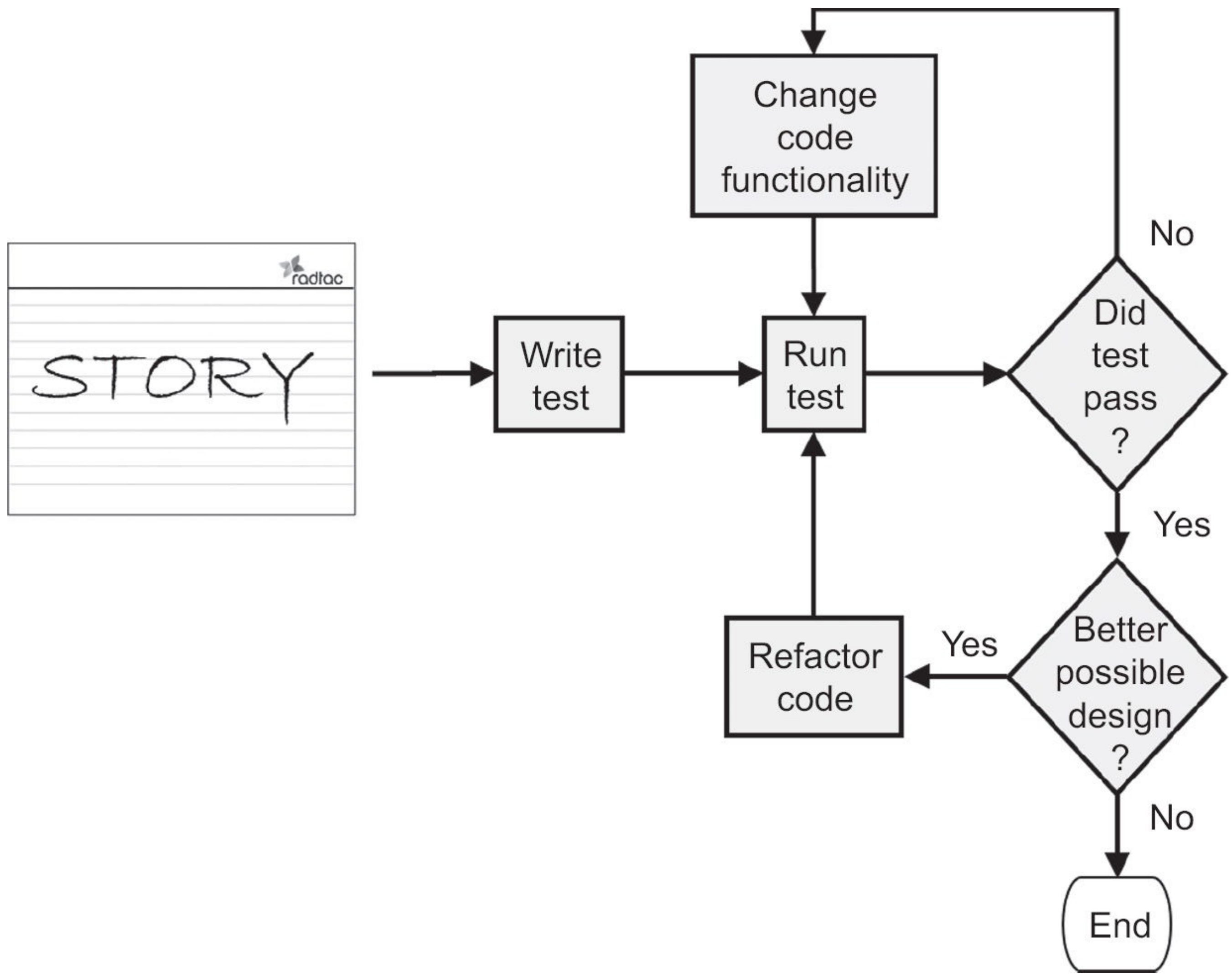
TFD validates that what has been built meets the story acceptance criteria (via the tests) agreed by the customer.

7.4.1.2 TDD (Test Driven Development)

Test Driven Development (Beck, 2002) is normally implemented at the unit or component testing level. In essence it is a style of TFD; the main difference is that TDD normally includes the Agile practice of ‘refactoring’ (see Section 8.10).

TDD validates that what has been built passes the tests and therefore meets the acceptance criteria upon which the tests are based, and that the design is appropriate with minimal technical debt. The TDD development cycle (sometimes known as ‘red-green-refactor’) is shown in Figure 7.6:

Figure 7.6 Test Driven Development



The benefits of TDD are numerous:

- Good design principles need to be followed to test code in isolation.
- The speed of the test–code cycle (enabled by automated testing tools) enables fast refactoring and therefore is a big enabler of emergent design (see Section 9.2).
- The focus is on the interface of the code.
- The unit test documents the expected behaviour of the code.
- The unit test is repeatable and can be automated.

In TDD, the team focuses on conditions in the test that could cause the code to fail. Once there are no more failure conditions, the development is said to be complete. Automated tests (see Section 8.10) give the team the confidence that the system operates as expected. As new stories are added, existing tests will quickly identify any unexpected issues. The focus in TDD is on design and ensuring that products are designed in a fit-for-purpose way.

7.4.1.3 ATDD (Acceptance Test Driven Development)

Acceptance Test Driven Development (Pugh, 2011) is very similar to TDD, although it is closer to user acceptance testing (UAT). While it is an effective approach to testing, many organisations prefer to use behaviour-driven development (see Section 7.4.1.4); therefore this book does not expand upon ATDD any further.

7.4.1.4 BDD (Behaviour Driven Development)

Behaviour Driven Development (North, 2006) focuses on scenario testing to make sure that the system behaves in the way the user expects it to behave. In this, BDD goes beyond simply delivering software that works and is designed well; it provides a very effective bridge between people with analytical, design, coding and testing skills because it encourages them to work together as a team rather than just passing documents to each other.

A detailed description of how to write story acceptance criteria related to BDD may be found in Section 7.1.2.

All of the practices described above rely on automated testing and effective continuous integration (see Section 8.10.2).

7.4.1.5 Specification by example

Specification by example (Adzic, 2010) and (Fowler, 2011) ensures that what is being created matches the customer's requirements and that testing is focused on the parts of the system that create the greatest business value. Gojko Adzic defines specification by example as

> … a set of process patterns that facilitate change in software products to ensure that the right product is delivered efficiently. When I say the right product, I mean software that delivers the required business effect or fulfills a business goal set by the customers or business users and it is flexible enough to be able to receive future improvements with a relatively flat cost of change.
>
> (Adzic, 2010)

There are a number of key elements to specification by example:

- deriving system scope from business goals that are clearly expressed and understood by the customer and stakeholders;
- specifying acceptance criteria for stories collaboratively between team, customer and stakeholder, as well as agreeing appropriate levels of testing detail;
- illustrating and agreeing requirements using examples (the team may add more detail in relation to edge cases where required);
- refining specifications throughout the lifetime of the product;
- automating validation without changing specifications;
- validating frequently that the product being built meets the specifications;
- evolving a documentation system that consists of simple-to-understand requirements and tests that prove them.